Structure and Interpretation of Computer Programs

Harold Abelson, Gerald and Julie Sussman

-- Chapter 1.1-1.2

Presented by Yan Yan
CSE 294, UCSD, April 13, 2011
The Idea of Computational Process

• Process
  – Abstraction of execution in computers.

• Data
  – Abstract things manipulated by processes.

• Program
  – A pattern of rules that directs the evolution of a process.

• About Clojure
Building Abstractions with Procedures

• Elements of Programming
  – Primitive expressions
  – Means of combinations
  – Means of abstractions

• Procedures and Processes
  – Recursive Processes and Iterative Processes
  – Recursive Procedures
The Elements of Programming

• Every powerful language should have the following three mechanisms to accomplish the goal of “combining simple ideas to form more complex ones”
  – Primitive expressions
    • the simplest entities the language is concerned with.
  – Means of combination
    • by which compound elements are built from simpler ones.
  – Means of abstraction
    • by which compound elements can be named and manipulated as units.
Primitive Expressions and Combinations

• Primitive Expressions
  – e.g. 5, 1.2, false
  – e.g. +, mod, or

• Combinations
  – e.g. (+ 10 5)
  – e.g. (not true)
Naming and Environment

• Naming
  – To identify a variable whose value is the object.
  – e.g. (def my-age 18)
    • Then we can refer to 18 by name: (+ 5 my-age)

• Environment
  – The interpreter needs to maintain the “memory” to keep track of the “name-object” pairs.
  – Such “memory” is called the environment.
Compound Procedures

• The abstraction technique by which a compound operation can be given a name and then referred to as a unit, e.g. -

  ( defn square [ z ]
       ( * z z )
  )

  ( defn square-sum [ x y ]
       ( + (square x) (square y))
  )

  ( defn f [ a ]
       ( square-sum (+ a 2) (- a 1))
  )
Evaluating a combination

\[( + (- 4 2) (/ 8 2) ) \]

\[( + 2 (/ 8 2) ) \]

\[( + 2 4) \]

6
Evaluating Compound Procedures

- Substitution for procedure application:

\[
f (5) \\
(square\text{-}sum \ (+ \ a \ 2) \ (- \ a \ 1)) \quad (\text{where } a = 5) \\
\quad \rightarrow \ (square\text{-}sum \ (+ \ 5 \ 2) \ (- \ 5 \ 1)) \\
(square\text{-}sum \ 7 \ (- \ 5 \ 1)) \\
(square\text{-}sum \ 7 \ 4) \\
(+ \ (square \ 7) \ (square \ 4)) \\
(+ \ (* \ 7 \ 7) \ (square \ 4)) \\
(+ \ 49 \ (square \ 4)) \\
(+ \ 49 \ (* \ 4 \ 4)) \\
(+ \ 49 \ 16) \\
65
\]
Applicative Order versus Normal Order

- Applicative order
  
  \[ f(5) \rightarrow \]
  
  \( (\text{square-sum}
    \begin{align*}
      (+ a 2) & (- a 1) \\
      (+ 5 2) & (- 5 1) \\
      7 & 4 \\
      (+ (\text{square } 7) (\text{square } 4)) \\
      (+ (* 7 7) (* 4 4)) \\
      (+ 49 16)
    \end{align*}
  ) \]

  \( + 65 \)

- Normal order
  
  \[ f(5) \rightarrow \]
  
  \( (\text{square-sum}
    \begin{align*}
      (+ a 2) & (- a 1) \\
      (+ 5 2) & (- 5 1) \\
      (+ (\text{square } 5 2) (\text{square } - 5 1)) \\
      (+ (* (+ 5 2) (+ 5 2)) (* (- 5 1) (- 5 1))) \\
      (+ (* 7 7) (* 4 4)) \\
      (+ 49 16)
    \end{align*}
  ) \]

  \( + 65 \)
An example from Chap. 4

• Test which order Clojure evaluation is using:

```clojure
(defn test-order [a b]
  (if true
    a
    b))
```

Test this:

```clojure
(test-order 1 (/ 1 0))
```
Conditional Expressions

• Case analysis

  (cond
   <predicate_1>  <expression_1>
   <predicate_2>  <expression_2>
   ...  ...
   <predicate_n>  <expression_n> )

• If :
  – a special case of cond (binary branch)
Procedure as Black-box Abstraction

• A procedure definition should be able to suppress detail.
  
  – e.g.

```
(defn compute-hypt [a b]
  (square-root (square-sum a b)))
```
Outline

• Elements of Programming
  – Primitive expressions
  – Means of combinations
  – Means of abstractions

• Procedures and Processes
  – Recursive Processes and Iterative Processes
  – Recursive Procedures
**Procedures and the Processes**

- A procedure is a pattern for specifying the evolution of a computational process.
- The process generated is the abstraction of program execution.
Linear Recursion

(defn factorial [n] ; compute n!
  (if (= n 1)
    1
    (* n (factorial (- n 1))))
Linear Recursion

(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
(defn factorial [n] (fact-iter 1 1 n)) ; compute n!

(defn fact-iter [product counter max-count]
  (if (> counter max-count)
    product
    (fact-iter (* counter product)
                (+ counter 1)
                max-count)))
Linear Iteration

(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720

Theoretically true, but, no actually.. (We’ll revisit Linear Iteration later.)
Recursive Procedure

• The procedure that refers to itself
  – When we describe a procedure as recursive, we are referring to the syntactic fact that the procedure definition refers to the procedure itself.

• Recursive processes and iterative processes can both be generated from recursive procedure.
Tree Recursion

• Fibonacci number:

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \]

\[
\text{Fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{otherwise}
\end{cases}
\]
Tree Recursion

(defn fib [n]
  (cond (= n 0) 0
        (= n 1) 1
        :else (+ (fib (- n 1))
                (fib (- n 2)))))

;Let’s compute (fib 5) --
Tree Recursion
To Linear Iteration?

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- 1. fixed number of state variables? yes
- 2. fixed rules of state transition? yes
- 3. end condition? yes

Then, we are ready to go --

```clojure
(defn fib [n]
  (fib-iter 0 1 n))

(defn fib-iter [curr next count]
  (if (= count 0)
    curr
    (fib-iter next (+ curr next) (- count 1)))))
```
Tail Recursion

- Defined recursively, but the recursion must come at the tail.

```clojure
(defn fib [n]
  (fib-iter 1 0 n))

(defn fib-iter [curr next count]
  (if (= count 0)
    curr
    (fib-iter next (+ curr next) (- count 1)))))
```
Tail Recursion

• JVM doesn’t support TCO automatically, so (fib 10000000) will lead to a stack overflow
• In Clojure, make tail calls explicit by using “recur”:

  (defn fib [n]  
    (fib-iter 1 0 n))

  (defn fib-iter [curr next count]  
    (if (= count 0)  
      curr  
      (recur next (+ curr next) (- count 1)))))
Summary

• Elements of Programming
  – Expressions and Combinations
  – Naming and Environments
  – Evaluations
  – Procedure as Black-box Abstractions

• Procedures and Processes
  – Recursion and Iteration
  – Tail Recursion
Q&A